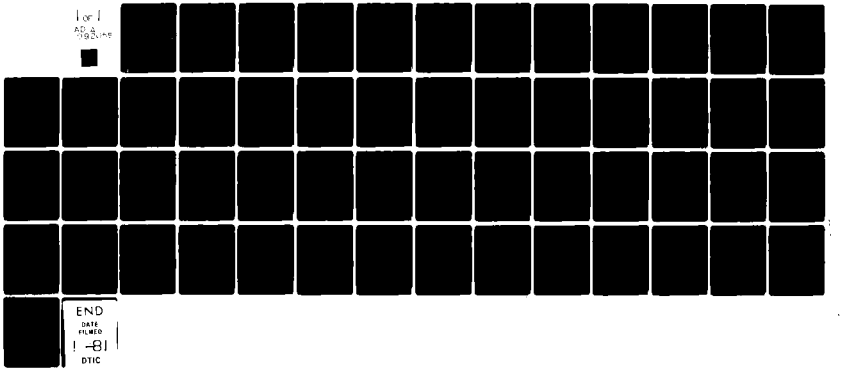
AD-A092 055 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/G 9/2
NETWORK MANAGEMENT RESEARCH.(U)
SEP 80 C V RAMAMOORTHY DAAG29-79-C-0171
UNCLASSIFIED ARO-16984.1-A-EL NL



LEVEL II

(12)

(6) NETWORK MANAGEMENT RESEARCH

(9) TECHNICAL REPORT

(16) C. V. Ramamoorthy

(11) September 1980

(12) 55

U. S. Army Research Office

(15) Grant DAAG29-79-C-0171

Electronics Research Laboratory

University of California

Berkeley, California 94720

THE VIEWS, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A092	3. RECIPIENT'S CATALOG NUMBER 055
4. TITLE (and Subtitle) Network Management Research		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL REPORT
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) C. V. Ramamoorthy		8. CONTRACT OR GRANT NUMBER(s) DAAG29-79-C-0171
9. PERFORMING ORGANIZATION NAME AND ADDRESS Electronics Research Laboratory University of California Berkeley, California 94720		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE September 1980
		13. NUMBER OF PAGES 53
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

shortest path algorithm, Petri nets, protocols, optimal reconfiguration strategies, interconnection networks, deadlocks, scheduling

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In our research in distributed data processing systems, we have developed design principles, algorithm concepts, feasibility criteria, and quantitative trade-offs that can be used in the creation of network management functions. In particular, we concentrated on the following five areas: (1) Routing Control and Relay Management; (2) Reconfiguration Control; (3) Interconnection Network Design; (4) Deadlock Detection; and (5) Scheduling. The results we have developed will provide guidelines and design laws for the systematic design and construction of distributed data processing systems such that the users' requirements are satisfied.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

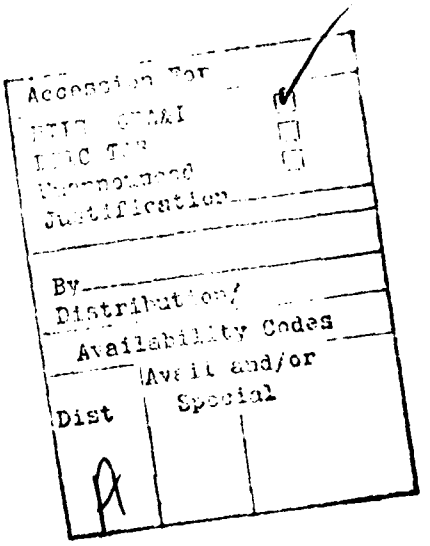
By using these results, we will be able to develop reliable, effective, modifiable networks with low cost and lead times.

Table of Contents

ABSTRACT

In our research in distributed data processing systems, we have developed design principles, algorithm concepts, feasibility criteria, and quantitative trade-offs that can be used in the creation of network management functions. In particular, we concentrated on the following five areas: i) Routing Control and Relay Management; ii) Reconfiguration Control; iii) Interconnection Network Design; iv) Deadlock Detection; and v) Scheduling. The results we have developed will provide guidelines and design laws for the systematic design and construction of distributed data processing systems such that the users' requirements are satisfied. By using these results, we will be able to develop reliable, effective, modifiable networks with low cost and lead times.

keywords: shortest path algorithm, Petri nets, protocols, optimal reconfiguration strategies, interconnection networks, deadlocks, scheduling



1. Introduction

Distributed Data Processing system is a collection of federation of computing elements whose concurrent and/or sequential operations solve a homogeneous problem. The overall objective of our research is to develop a design methodology and to establish a basis for design theory for the development of distributed processing systems. The desired end result of this research are design principles, feasibility criteria, algorithm concepts, and quantitative trade-offs that can be used in the design of network management functions for large tactical distributed data networks. To achieve these, our research has concentrated on the following five areas: i) Routing Control and Relay Management; ii) Reconfiguration Control; iii) Interconnection Network Design; iv) Deadlock Detection; and v) Scheduling.

The goal of routing control and relay management is to create network management algorithms thereby automatic designation of relaying responsibility can be accomplished cooperatively by neighboring nodes. We have developed a shortest path routing algorithm with expected execution time $O(\sqrt{n} \log n)$, where n is the number of nodes in the network. This algorithm is by far the fastest algorithm (w.r.t. average execution time) in the literature. We have also used Petri net models to describe and design protocols.

In the area of reconfiguration control, we have developed design strategies for maximizing the operational reliability and upgrading the system fault tolerance. We have investigated the optimal reconfiguration strategies which maximize the global performance of the dynamic network when failures occur. In addition, we have developed four different models for the dynamic reconfigurable system, which are different in complexity and generality. By using Petri net models, we have developed a reconfiguration procedure to detect and isolate potential faults in the protocol design, to

identify the associated recovery mechanisms and to choose the most effective one.

To design the network with continuously changing configuration and a large amount of message transfers, we have proposed to develop associative network management concepts to maximize the processing rate at individual nodes and to minimize the overhead in communication bandwidth. We have classified the existing interconnection networks with respect to the characteristic of physical connections and communication paths in the networks. The characteristic of the interconnection network is then revealed by relating the similarities and distinguishing the differences to the other existing interconnection network.

Deadlock detection is part of system verification. In a distributed allocation environment, deadlock detection presents some additional problems arising from message delays as compared to detection in a centralized system. We have developed a discipling of requesting resources and releasing them which eliminates these problems instead of increasing the time and space overhead involved in Ho's algorithm [HO 79]. We have also developed a truly distributed algorithm for detection which involves much lower overhead than one presented in [MEN 79].

To develop a schedule of sequencing the execution of the parallel task in real network environment, we have to evaluate the overhead effect of the communication delay time. The aim is to develop techniques that a program can be executed in the least with the consideration of the overhead involved in the parallel execution of tasks. We have used deterministic graph models to find the optimal schedules for the minimum execution time of the program with a specified number of processors.

In the following sections, the results we have obtained in each area are presented in detail.

2. Routing Control

2.1 Routing Algorithm for the Dynamic Interconnection Network

We have developed a 2-node shortest path algorithm with fastest average execution time among all the existing shortest path algorithms [MA 80]. Based on this algorithm, we will develop an efficient routing strategy for the dynamic network. The assumptions that we have made on the algorithm are: 1) The lengths of the edges in the network are drawn independently from a common exponential distribution. 2) There exists a preprocessing phase that sorts all the in-edges and out-edges with respect to edge lengths for each node in the network. The average execution time of the algorithm is O(5n log n) for a network with n nodes.

The proposed algorithm is a modified version of the Dijkstra's shortest path algorithm with bidirectional search. The idea of the algorithm is briefly discussed below by first reviewing the Dijkstra's algorithm, then the conventional bidirectional algorithm, and finally, the proposed algorithm is presented.

2.1.1 Dijkstra's Shortest Path Algorithm

Let $\pi(v)$ denote the current shortest distance from s to v. Let S be the set of nodes whose minimum distance from s is known. Let pred(v) denote the immediate predecessor of v in the shortest path from s to t and $pred^k(v)$ denote pred(pred...(v))(k times). Dijkstra's algorithm can then be described as follows:

Dijkstra's Shortest Path Algorithm:

```
begin
  π*(s) = 0;
  S = {s};
  for (v∈V-{s}) do
    π*(v) = d(s,v);
  while (t∉S) do
  begin
    u = the node in V-S such that π*(u) = min {π*(v)};
                                          v∈V-S
```

```
    S = S∪u;
    for (v∈V-S) do
      if (π*(v)>π*(u)+d(u,v)) then
      begin
          pred(v) = u;
          π*(v) = π*(u)+d(u,v);
      end;
  end;
  The shortest path is the path (s,pred^k(t),...,pred(t),t)
end.
```

The execution time of the above algorithm is dominated by the following two operations:

1. Selecting a node v in V-S to minimize $\pi^*(v)$;
2. Updating $\pi^*(v)$, for v in V-S.

An $O(|E|\log|E|)$ execution time can be achieved if a data structure called a priority queue is used to support the above operations. A priority queue represents a set of items of the form $<x,v>$, where x is an arbitrary data object and v is a real number called the value of the item; it supports the operations of insertion, and deletion of the item with minimum value, in time $O(\log N)$, where N is the number of items in the priority queue. Let Q denote the priority queve. Using this data structure, Dijkstra's algorithm is stated as follows:

Dijkstra's algorithm with priority queue:

```
begin
  Q = φ;
  S = {s};
  π*(s) = 0;
  for (v∈out(s)) do
  begin
    insert <(s,v),π(s)+d(s,v)>into Q;
  end;
  while (t∉S) do
  begin
    delete the minimum item <(u,v),v>from Q;
    if (v∉S) then
    begin
        S = S∪{v};
      π*(v) =v;
```

```
      pred(v) = u;
      for (w out(v) and w∉S) do
          insert <(v,w),π(v)+d(v,w)> into Q;
    end;
  end;
  The shortest path is the path (s,pred^k(t),...,pred(t),t);
end.
```

2.1.2 The Bidirectional Shortest Path Algorithm

The idea of the bidirectional shortest path algorithm is to apply Dijkstra's shortest path procedure from s and t simultaneously. We first define the following notations:

FWS Dijkstra's shortest path procedure fanned out from s

BWS Dijkstra's shortest path procedure fanned out from t

$\pi_f(v)$ forward label of node v, which is the actual shortest distance from s to v

$\pi_b(v)$ backward label node of v, which is the actual shortest distance from v to t

S set of nodes whose minimum distanct from s is known

T set of nodes whose minimum distanct to t is known

$\tilde{S}$ set of nodes reached from S by one edge but not in S

$\tilde{T}$ set of nodes reached from T by one edge but not in T

$\overline{S}$ complement set of S

$\overline{T}$ complement set of T

Using these notations, the bidirectional shortest path algorithm is given as follows:

Bidirectional Shortest Path Algorithm with priority queues

```
Procedure FWS()
 /* procedure FWS returns the next node v that should enter S */
begin
  repeat
    delete the minimum item <(u,v),v> from Q_f;
  until (v∉S);
  pred(v) = u;
```

```
  π_f(v) = ν;
  for (w∈out(v) and w∉S) do
    insert <(v,w),π_f(v)+d(v,w)> into Q_f;
  return (v);
end;

Procedure BWS()
 /* procedure BWS returns the next node v that should enter T */
begin
  repeat
    delete the minimum item <(v,u),ν> from Q_b;
  until (v∉T);
  suc(v) = u;
  π_b(v) = ν;
  for (w∈in(v) and w∉T) do
    insert <(w,v),π_b(v)+d(w,u)> into Q_b;
  return (v);
end;

/* main program */
Main()
begin
  Q_f = φ;
  Q_b = φ;
  S = {s};
  T = {t};
  π_f(s) = 0;
  π_b(t) = 0;
  for v∈out(s) do
    insert <(s,v),π_f(s)+d(s,v)> into Q_f;
  for v∈in(t) do
    insert <(v,t),π_b(t)+d(v,t)> into Q_b;
  disjoint = true;
  while (disjoint) do
  begin
    w = FWS();
    S = S∪{w};
    if (w∈T) then disjoing = false;
    else
    begin
        w = BWS()
        T = T∪{w};
        if w∈S then disjoint = false;
    end;
  end;
  determine π_b(v) for all v∈S∩T̃
  w = node ∈S∩(T∪T̃) such that
       π_f(w)+π_b(w) =   min   {π_f(v)+π_b(v)}
                       v∈S∩(T∪T̃)
  the shortest path is (s,pred^i(w),..., pred(w), w,suc(w),...,suc^j(w),t);
end.
```

Although in the worst case, the computation required by the above

algorithm exceeds that for the Dijkstra's algorithm, it has been shown by simulation that the average behavior of the bidirectional shortest path algorithm is better. In the next section, we will present a variant of the bidirectional shortest path algorithm which has an $O(\sqrt{|V|}\log|V|)$ expected running time under certain assumptions.

2.1.3 Proposed Bidirectional Shortest Path Algorithm

Our proposed algorithm is divided into two phases: Phase 1 begins with $S = \{s\}$, $T = \{t\}$. It performs FWS and BWS to expand S and T by one node at a time alternately until a node in $S \cup T$ is found. Phase 2 consists of the rest of the algorithm. Our algorithm modifies the conventional bidirectional algorithm so that it performs very efficiently on the average while the worst case execution time remains $O(|E|\log|E|)$. The algorithm is modified in two aspects: First, the insertion and deletion operations on the priority queues are modified to reduce the number of such operations. Second, after a node $v \in S \cap T$ is found, we use an elimination process to establish the node g, where $g \in S \cap (T \cup \tilde{T})$ and

$$\pi_f(g)+\pi_b(g) = \min_{v \in S \cap (T \cup \tilde{T})} \{\pi_f(v)+\pi_b(v)\}$$

We call the node g the target node. The details of the modifications are explained in the following subsections:

2.1.3.1 The priority queues:

When executed on typical examples the shortest path algorithms we have presented spend most of their time pouring items into the priority queues, while very few of these items are ever selected from the priority queues. A trick similar to the one presented in [1] is used which reduces drastically the number of priority queue insertions. Recall that the operations we need to perform on Q_f in the FWS are:

1. insertion

   for ($w \in out(v)$ and $w \notin S$) do

     insert $<(v,w), \pi_f(v)+d(v,w)>$ into Q_f.

2. deletion

   delete the item $<(v,w),v>$ with minimum value of v from Q_f.

3. $Q_f = \phi$

The proposed bidirectional shortest path algorithm assumes a preprocessing phase in which, for each node v, two lists, namely, inlist(v) and outlist(v), are formed, where inlist(v) consists of all ordered pairs (u,v) such that $(u,v) \in E$, sorted in increasing order of d(u,v), and outlist(v) consists of all ordered pair (v,u) such that $(v,u) \in E$, sorted in increasing order of d(v,u). Let Q'_f [Q'_b] be the priority queue modified by our trick, which is used in the FWS[BWS]. The relation of Q_f to Q'_f is that $<v,u> \in Q_f$ if and only if $(v,w) \in Q'_f$, wehre w = u, or w is a predecessor of u in out(v). The above operations on Q'_f modified as follows:

1. insertion

   (v,w) = first element in out(v) such that $w \notin S$;

   if (v,w) exists then

     insert the item $<(v,w), \pi_f(v)+d(v,w)>$ into Q'_f.

2. deletion

   delete the item $<(v,w),v>$ with minimum value from Q'_f;

   (v,u) = first successor of (v,w) in ouz(v) such that $u \notin S$;

   if ((v,u) exists) then

     insert the item $<(v,u), \pi_f(v)+d(v,u)>$ into Q'_f.

3. $Q'_f = \phi$

Q'_b and the associated operations are modified similarly. With this modification, we shall show in the next section that the expected number

of insertions and deletions performed in $O(\sqrt{|V|})$.

2.1.3.2 Establishment of the target node g

Let S'[T'] be the set of nodes in S[T] at the end of phase 1. As mentioned before, once a node in $S \cap T$ is found, the target node g is the one in $S \cap (T \cup \tilde{T})$ which satisfies

$$\pi_f(g)+\pi_b(g) = \min_{v \in S \cap (T \cup \tilde{T})} \{\pi_f(v)+\pi_b(v)\}$$

In phase 2, instead of finding the target node by

1. determining the set $S' \cap \tilde{T}'$,
2. computing $\pi_b(v)$, for $v \in S' \cap \tilde{T}'$,
3. finding the target node by minimizing $\pi_f(v)+\pi_b(v)$, for $v \in S' \cap (T' \cup \tilde{T}')$, our algorithm uses a different approach that reduces the amount of computation on the average. It finds the target node g using the following procedures:
   1. Choose a candidate set containing all the nodes which are possible candidates for the target node. Since node $g \in S' \cap (T \cup \tilde{T}')$ implies that $g \in S' \cup T'$, the set $S' \cup T'$ is used as the initial candidate set.
   2. Computer the lower bounds $\pi'_f(v)$ on $\pi_f(v)$, for $v \in T'$, lower bound $\pi'_b(v)$ on $\pi_b(v)$, for $v \in S'$, and upper bound B(s,t) on the length of the shortest path.
   3. Find the target node by eliminating the nodes from the candidate set using the above upper and lower bounds. The way to compute the bounds and zhe elimination rules are explained in the following:

2.1.3.2.1 The Computation of Bounds

Let S*[T*] denote the set of nodes in S'[T'] which have not been eliminated. A node v is said to be selected by the FWS[BWS] if $\pi_f(v)[\pi_b(v)]$ has been determined by the FWS[BWS]. In phase 2, if

1. The FWS[BWS] selects a node $w \notin S$[$w \notin T$], then $\pi_f(w)$[$\pi_b(w)$] can be used as the lower bound $\pi'_f(v)$[$\pi'_b(v)$], for $v \in T^*$[$v \in S^*$].
2. The FWS[BWS] selects a node $w \in T'$[S'], then $\pi_f(w) + \pi_b(w)$ can be used as the upper bound $B(s,t)$.

The bounds $\pi'_f(v)$, for $v \in T^*$, $\pi'_b(v)$, for $v \in S^*$, and $B(s,t)$ are updated and refined if possible when the FWS and BWS proceed in phase 2.

2.1.3.2.2 The Elimination Rules

A node v is called the tentative target node if both the valued of $\pi_f(v)$ and $\pi_b(v)$ have been determined and $\pi_f(v) + \pi_b(v) = B(s,t)$. A node v is eliminated from the candidate set if either

1. It becomes the tentative target node.
2. It becomes ineligible for the target node, that is, if

$$\pi_f(v) + \pi'_b(v) \geq B(s,t), \text{ for } v \in S^*, \text{ or}$$

$$\pi'_f(v) + \pi_b(v) \geq B(s,t), \text{ for } v \in T,$$

Other important features of phase 2 are that when an item $<(v,u),v>$[$<(u,v),v>$] is deleted from Q'_f[Q'_b], no new item of the form $<(u,w), \pi_f(u) + d(u,w)>$ [$<(w,u), \pi_b(u) + d(w,u)>$] where $w \in out(u)$[$w \in in(u)$] is inserted into Q'_f[Q'_b]. Moreover, when a node v is eliminated from S^*[T^*], the item $<(v,u),v>$ [$<u,v),v>$] is deleted from Q'_f[Q'_b] if the item exists. It is easy to verify that these two points do not affect the correctness of the algorithm, since all the nodes in the shortest path are contained in $S' \cup T'$. However, these two points reduce the number of insertion and deletions in the priority queues, and enable us to achieve the $O(\sqrt{n} \log(n))$ expected running time, where n denote the size of set V. Let z be the node in $S' \cap T'$ found by phase 1. Phase 2 begins

with the candidate set equal to S'∪T' - {z}. It performs FWS until a node in T* is eliminated, then it switch to BWS until a node in S* is eliminated. Only one node is eliminated at a time in order to simplify the analysis. Phase 2 ends when either S* or T* becomes empty. The details of the algorithm are given as follows:

Proposed Bidirectional Shortest Path Algorithm:

```
Procedure FWS()
begin
  repeat
  begin
    delete the minimum item <(u,v),ν> from Q'f;
    (u,w) = first successor of (u,v) in out(u) such that w∉S;
    if ((u,w) exist) then
        insert <(u,w),πf(u)+d(u,w)> into Q'f;
  end;
  until (v∉S);
  πf(v) =ν;
  if (phase 1) then
  begin
    (v,w) = first element in out(v) such that w∉S;
    insert <(v,w),πf(v)+d(v,w)> into Q'f;
    pred(v) = u;
  end;
  return (v);
end;

Procedure BWS()
begin
  repeat
  begin
    delete the minimum item <(v,u),ν> from Q'b;
    (w,u) = first successor of (v,u) in in(u) such that w∉T;
    if ((w,u) exists) then
        insert <(w,u),πb(u)+d(w,u)> into Q'b;
  end;
  until (v∉T);
  πb(v) = ;
  if (phase 1) then
  begin
    (w,v) = first element in in(v) such that w∉T;
    insert <(w,v),πb(v)+d(w,v)> into Q'b;
    suc(v) = u;
  end;

/* main program */
Main()
begin
```

```
Q'f = φ;
Q'b = φ;
S' = φ;
T' = φ;
S = {s};
T = {t};
πf(s) = 0;
πb(t) = 0;
(s,v) = first element in out(s);
insert <(s,v),πf(s)+d(s,v)> into Q'f;
(v,t) = first element in in(t);
insert <(v,t),πb(t)+d(v,t)> into Q'b;
/* phase 1 begins */
disjoint = true;
while (disjoint) do
begin
  w = FWS();
  S = S∪{w};
  if (w∈T) then disjoint = false;
  else
  begin
      w = BWS();
      T = T∪{w};
      if (w∈S) then disjoint = false;
  end;
end;
/* phase 1 ends */
S* = S*-{w};
T* = T*-{w};
delete the item <(w,v),ν> from Q'f if it exists;
delete the item <(v,w),ν> from Q'b if it exists;
B = πf(w)+πb(w);
target = w;
sort the nodes v in S* according to increasing value of πf(v);
sort the nodes v in T* according to increasing value of πb(v);
/* phase 2 begins */
while (|S*|>0 and |T*|>0) do
begin
  elim = false;
  while (not elim) do
  begin
      w = FWS();
      S = S∪{w};
      if (w∈T*) then
      begin
        T* = T*-{w};
        elim = true;
        delete the item <(v,w),ν> from Q'b if it exists;
        if πf(w)+πb(w) < B then
        begin
          B = πf(w)+πb(w);
          target = w;
        end;
      end
```

```
          else
          begin
            temp = B-π_f(w);
            r = the last node in T*;
            if (π_b(r)≥temp) then
          begin
              T* = T*-{r};
              delete the item <(r,v),ν> from Q'_b if it exists;
              elim = true;
            end;
          end;
      end;
      if (|T sup *|>0) then
      begin
          elim = false;
          while (not elim) do
          begin
            w = BWS();
            T = T∪{w};
            if (w∈S*) then
            begin
              S* = S*-{w};
              elim = true;
              delete the item <(v,w),ν> from Q_f if it exists;
              if (π_b(w)+π_b(w) < B) then
              begin
                  B = π_f(w)+π_b(w);
                  target = w;
              end;
            end
          else
            begin
              temp = B-π_b(w);
              r = the last node in S*;
              if (π_f(r)≥temp) then
              begin
                S* = S*-{r}
                delete the item <(r,v),ν> from Q'_f if it exist;
                elim = true;
              end;
            end;
          end;
      end;
    end;
    /* phase 2 ends */
    the shortest path is (s,pred^i(target),...,pred(target),target,suc(target),
    ...suc^j(target),t);
end.
```

2.1.4 The Analysis of the Algorithm

The Analysis of the Algorithm is very complicate due to the dependency effect among the random variables, as well as the complicate distribution

function on the random variables which are used in the analysis. The technique that we used to overcome the difficulties is the concept of "stochastic dominance" among distribution functions. The detail of the analysis is not presented here due to its length, and only the main lemmas and theorems that we have established are summarized in the following:

Lemma 1: E (number of edges inspected in phase 1) = $O(\sqrt{n})$.

Lemma 2: E (number of edges inspected in phase 2) = $O(\sqrt{n})$.

Theorem: E (execution time of the algorithm) = $O(\sqrt{n} \log n)$.

pf: Since each inspection operation involves at most one insertion and one deletion from the priority queue, therefore the result follows.

2.2 Design of Protocols

A communication protocol is a set of rules which control the exchange of information among processors in a communication network. The design and implementation of protocols strongly affects the performance of a computer network.

In a computer network, protocols should have the following functions:

1. Synchronizing the logical interactions between processors to ensure the communication is deadlock-free,
2. Detecting the errors in the received message, the loss of control and data message, and effecting retransmission or time out procedure when needed to ensure that the communication is error tolerant,
3. Controlling the flow through the network to ensure that the communication is bounded, i.e. regulating the number of messages sent to each processor such that the queues of the processor or the link can accomodate all of the simultaneously received messages.

These issues have not been handled well in the past because of the lack of tools for describing, defining, implementing and measuring the

efficiency of protocols. After studying several models, we find that Petri nets are very suitable models to describe communication protocols because there are many corresponding properties between Petri nets and protocols. In section 3.2, we will describe a systematic procedure to design a sound protocol from a very primitive one.

3. Reconfiguration Control

3.1 Optimal Reconfiguration Strategies

The motivation of investigating optimal reconfiguration strategies is that at any given failed state, the system may have different ways to reconfigure into different subsequent functional states in which the system performance such as reliability, availability and throughput are varied. Therefore, the resulting performance of the entire system depends on the particular reconfiguration strategy used by the system at each failed state. Hence, it is important to find the optimal reconfiguration strategies so that the overall system performance can be optimized.

3.1.1 Previous Work

Troy [TRO77] developed a simple model to find the optimal reconfiguration strategy for a dynamic reconfigurable system. The system he modeled consists of only one type of physical resource with n identical copies. The resource is used to execute two types of processes: vital X and nonvital Y. Initially, the system allocates x copies of the resource to X and y to Y. The failure of the copies in X does not lead to fall-off in system performance as long as $x > 0$ (they are used as redundant resources), but $x = 0$ leads to system failure, while every failure of the copies in Y leads to a fall off in system performance, but $y = 0$ does not lead to system failure. The objectives of the system are: to keep the system from failure and to maintain the system performance as high as possible. Under these assumptions, Troy found that the optimal reconfiguration strategy is to switch in a copy of the resource from those allocated to Y into X whenever $x = 1$ and $y > 0$.

The limitations and drawbacks of this model are:

1. The model assumes only one type of resource in the system and cannot be generalized to systems with multiple types of resource. This make

the model inapplicable to real systems.

2. The objective function consists of only two parameters, namely, availability and performance. It cannot be generalized to include other parameters, like reconfiguration overhead, etc.
3. The vital and non-vital characteristics of the processes in the system must be satisfied, which may be restrictive in real applications.
4. The failure rate of the resource is state independent. This assumption may not be realistic. For example, for a system with limited amount of resources, deadlock occurrence may be more frequent when a large number of users are requesting services from the system.

Helviks [HEL78] addresses the problem of finding the optimal reconfiguration strategies from a different point of view. While Troy assumes that the diagnostic utilities in the system are fault-free, Helvick considers the situation where the diagnostic utilities may not be perfect, e.g., they may be unable to locate a fault to a specific unit, or classify a unit fault-free when it is faulty and vice versa. He assumes that the optimal reconfiguration strategy for each failure state is available in advance, and the "regret" $<\sigma_i, a_j>$ which is the loss of using a non-optimal strategy a_j in failure state σ_i is also available. Since the diagnostic utilities may not be perfect, the system is unsure of the state it is in even though the diagnostic utilities supply the corresponding information. Under this assumption, Helvicks suggests an algorithm which gives the "optimal" strategy for each diagnosis result, in the sense that the strategy minimizes the expected regret.

This model is interesting and useful in its own right. However, it is of little value unless the more fundamental problem of finding the optimal reconfiguration strategies has been solved.

3.1.2 The Proposed Models (stochastic models)

The guidelines in developing our stochastic models are:

1. The assumptions on the system must not be restrictive so that the models are applicable to real systems.
2. The objective function of the system should be very general so that any (finite) number of parameters can be included, e.g., reliability, availability, cost, etc.
3. We are interested in the global optimal reconfiguration strategies.

While in the area of designing the algorithms, which find the optimal strategies, we focussed our attention on the practicality of the algorithm; that is, the complexity of the algorithm is analyzed such that unless the problem is NP-complete, we insisted in finding a polynomial running time algorithm for the model. On the other hand, for the NP-complete problem, we investigated good heuristic with fast running time and small deviations. from the optimal solution.

Moreover, we restricted our attention to finding those algorithms which yield nonrandomized strategies. A strategy is called randomized if at any given state, it chooses an action θ_i with some probability P_i, where θ_i is an action in the action space, and a strategy is called nonrandomized if at any given state, it always chooses a fixed action θ_i. Although from the algorithm point of view, it is easier to develop a fast algorithm that yields the optimal randomized strategies, it is impractical even if not infeasible to use the randomized strategies in a real system, due to the fact that the number of randomized reconfiguration strategies that the system needs to implement is much larger than in the nonrandomized case. Further, use of randomized strategies also makes the system behavior nondeterministic, which is undesirable.

The common assumptions made by the models that we have developed are:

1. The system can have multiple number of resources with multiple copies.
2. In any given state, the system can undergo different number nodes of failure, which lead the sytem into different subsequent failure states. The failure state that the system entered is chosen according to probability P_{ij} where i is the current functional state, and j is the subsequent failure state.
3. In any failure state j the system may reconfigure itself into different functional states, according to reconfiguration strategies a_j^1, a_j^2,..., a_j^k, available to that state. However, if there is no reconfiguration strategy available, then this failure state is called a dead state, and the system failed completely.

We have developed four models which are different in the degree of generality and complexity. The detail of each model is explained in the following:

(i) Model I

In this model, the parameters that the system wants to optimize are combined into a reward function, which can be any general functions of the parameters. There are numberous studies on the design of such functions based on decision theory [KEE76]. For each reconfiguration strategy A_j^h, we assume that the corresponding reward is available which is a measure of the "goodness" of this strategy towards the objective of the system. For example, if availability is the only concern of the system, then the expected transition time from a functional state to the next failure state can be used as the reward for the strategy which leads to this functional state. The objective of the system is to optimize the expected sum of rewards, starting from the initial state to the dead state. The problem can be formally stated as follows:

$\max E [\sum_{i=0}^{m} R_i]$, where 0 = initial state

m = dead state

R_i = reward in state i.

The approach in finding the optimal reconfiguration strategies is by dynamic programming, yielding an algorithm with execution time O(NMA), where N is the number of states, M is the largest number of failures the system can survive, starting from the initial state to the dead state, and A is the total number of strategies.

(ii) Model II

This model differs from the first one in the following aspect: Instead of compressing all the performance measures into a reward function, the objective function consists of multiple reward functions. The objective is to maximize a particular reward function, subject to the constraints that the other reward functions less than some prescribed requirements. For example, the reconfiguration strategy may maximize the expected availability of the system, such that the expected total reconfiguration cost is less than a prescribed value, and the expected total throughput is greater than another prescribed value. This problem can be mathematically stated as follows:

$$\max E [\sum_{i=0}^{m} R_i]$$

$$\text{s.t. } E [\sum_{i=0}^{m} R_h^i] \leq \ell_h, \qquad h=2,\ldots,k$$

where

R_i: the i^{th} reward function

ℓ_i: prescribe requirement on the i^{th} reward function

This model is a generalization of model I. Moreover, this model gives a clearer picture of the objectives of the system, and helps the designer to express the system requirements more easily. The complexity of this problem is given by the following theorem.

Theorem 1: The problem of finding optimal reconfiguration strategies for model II is NP-complete.

proof: The proof is based on the reducibility of the 0-1 knapsack problem to this problem.

Theorem II: Pseudo-polynomial time algorithm exists for this problem with running time $O(\prod_{i=1}^{k} \ell_i \; NMA)$.

An algorithm is called a polynomial time algorithm if its running time is bounded by the number of bits that is needed to encode the encode. Since it only takes $\log(\ell_i)$ bits to encode each ℓ_i value, the algorithm with running time $O(\sum_{i=1}^{k} \ell_i \; NMA)$ is called pseudo-polynomial time algorithm. For most of the practical application, the ℓ_i's are bounded value (i.e. less than some constant) and the above algorithm will have $O(NMA)$ running time, and thus can be considered as a polynomial time algorithm.

At this stage, we conjecture that fully polynomial time approximation algorithm exists for this problem. An algorithm is called fully polynomial time approximation algorithm if for any E, wehre $0 < E < 1$, if P^* is the optimal solution, and P is the solution produced by the algorithm, then $P^* - P \leq E^*$. Moreover, the running time of the algorithm is bounded by the length of the encoded input and $\frac{1}{E}$. One of our current research areas is to investigate the design of the fully polynomial time approximation algorithm.

(iii) Model III

This model is a direct generalization of Model II. The objective of this model is to maximize a particular reward function, subject to the constraints that the other reward function satisfy some prescribed requirements, that is,

$$\max E\left[\sum_{i=0}^{m} R_i\right]$$

$$\text{s.t.} \quad E\left[\sum_{i=0}^{m} R^i\right] \leq \ell_h, \qquad h=2,\ldots,k_1$$

$$E\left[\sum_{i=0}^{m} R_h^i\right] \leq \ell_h, \qquad h=k_1+1,\ldots,k_2$$

We have proved the following theorem for this problem.

Theorem 3: The problem of finding the optimal reconfiguration strategies is strongly NP-complete.

A problem is strongly NP-complete means that a pseudo-polynomial time algorithm cannot be found unless P = NP.

(iv) Model IV

This model allows repairs on the failed resources to be done before the system enters the failed state. In other words, if we represent the system by a graph, the former stochastic models is an acylic digraph (the number of resources decreases in each transition) and this model is a digraph with any arbitrary number of cycle. Mathematically speaking, the former models deal with the finite horizon problem while this model address the infinite horizon problem. In this system, it is not meaningful to maximize the total expected measure since the sum of measure of any strategy may possibly be infinite. Instead, we find the optimal strategies which maximize the total expected discount measure in a state. The problem

is formulated as follows:

$$\max E\left[\sum_{i=0}^{m} \alpha^i R_i\right]$$

$$\text{s.t. } E\left[\sum_{i=0}^{m} \alpha^i R_i\right] \geq \ell_i, \qquad h=1,\ldots,k$$

where

α: discount factor.

We have proven that polynomial time algorithm exists for this problem. The techniques we used are the Markov decision process and Russian's Linear Programming Algorithm.

3.2 Reconfiguration Using Petri Net Models

Petri nets are a graph theoretical model of information and control flow. These are especially useful in systems that exhibit asynchronous and concurrent activities.

3.2.1 The Liveness Issue

One of the important properties of Petri nets is the "liveness" of the Petri net. A Petri net is said to be live if there always exists a firing sequence to fire each transition in the net. By guaranteeing that the Petri net is live, the system is guaranteed to be deadlock-free.

In order to study dynamic reconfigurability using the Petri net model, the first issue that has to be tackled is whether the Petri net is live on the occurrence of a fault, because this is essential for the reconfiguration strategy to operate.

The occurrence of fault in the Petri net can be two types - failure of a transition to occur or the illegal occurrence of a transition. The former is modelled by the loss of a token from a place, and the latter by the illegal generation of a token.

It is obvious that if the type of failure that occurs is the gain of a token, the Petri net that is live is still going to remain live, because all the transitions that were enabled before still remain enabled and some new transitions become enabled in addition. So the problem to be considered is whether the Petri net remains live on the loss of a token at a given place.

3.2.2 Reconfigurable Petri Nets

To study the above problem, a State Transition Graph is defined. A State Transition Graph (STG) is a graph that shows the execution of a Petri net, each state being one of the markings of the net. Another useful concept is the Event State Transition Graph (ESTG). We define the ESTG for the

event of a failure as the STG that corresponds to the occurrence of a failure in the form of a loss or gain of a token.

Fig. 3.1 is the Petri net model of the exchange of messages between a sender and a receiver. The STG corresponding to the execution of the Petri net has been shown in Fig. 3.2. One possible error that is quite common in such systems is the loss of message M, which is represented by the loss of a token at place M. Fig. 3.3 shows the ESTG corresponding to this failure.

We have derived the necessary and sufficient conditions for the reconfigurability of a Petri net on the occurrence of a failure.

Necessary and Sufficient Conditions for Reconfigurability

A Petri net P is recoverable from a failure F if and only if the ESTG of P for the failure F has the following properties:

1. The number of illegal states is finite.
2. There are no final illegal states.
3. There are no loops containing only illegal states.

A Petri net is reconfigurable if and only if it satisfies the conditions given above. In Fig. 3.3, there is a terminal illegal state WR in the ESTG and hance the Petri net in Fig. 3.1 is not recoverable.

In order to effect a recovery, somehow the terminal state WR must be detected and the system be put back in a legal state. This can be done in one of three different ways.

1. Using W to roll back the system to the state XR.
2. Using R to roll the system forward to state WG.
3. Using both W and R to roll the system back to state S.

Since the recovery mechanisms in such cases is usually time-dependent (eg. time-outs and retransmissions), some concept of time has to be inherent

in the model for it to represent such recovery mechanisms. This directly leads to the concept of Extented Twin Timed Petri Nets.

Extended Twin Timed Petri Nets (ETTPNs) are Petri nets where each transition t_i has two numbers r_i' and r_i'' defined as follows:

1. r_i' is the minimum time that must elapse after t_i becomes firable, after which it can actually fire.
2. r_i'' is the maximum time that can elapse after t_i becomes firable, before which it must fire.

Armed with this extended concept of Petri nets we can design dynamically reconfigurable systems.

In our example, suppose the first recovery strategy is chosen. This is reflected in the Petri net by the additional transition t_7 as shown in Fig. 3.4, the corresponding ESTG being Fig. 3.5. Transition t_7 can be physically interpreted as the time-out mechanism which will repeatedly send old messages until an ACK or a NAK signal is received.

Now we have to ensure that the transition t_7 does not fire during the normal operation of the Petri net. This would mean that the maximum time that it would take for the token in W to be normally consumed should be less than the minimum time that it would take for t_7 to fire. Mathematically,

$$r_7' > r_3'' + r_4'' + r_5''.$$

Thus, dynamic reconfigurability is achieved in this model without any degration in the performance of the system.

As a further illustration of this reconfiguration procedure, let us consider another common type of failure in such systems, namely the failure of the achknowledge signal to reach the sender. This is represented by loss of a token at A. The ESTG for this is given in Fig. 3.6. As can be seen from the figure, the transition t_7 will keep firing, generating infinitely many illegal states, prohibiting recovery.

This can be remedied by adding an additional transition t_8. This is

physically interpreted as saying that when a message arrives in the state where the receiver has already sent an acknowledge signal, the new message is known to be a repetition of the old message, and hance the system goes back to the state of having received the massage. The Petri net and the ESTG for this version are shown in Figs. 3.7 and 3.8, respectively. However, a time constraint has to be set so t_7 does not fire in the state WAK, but t_8 does. That is,

$$r_7' > r_8'' + r_4'' + r_5''.$$

By these procedures, we have developed the stop-and-wait Automatic Repeat Request (ARQ) communication protocol from our primitive protocol in Fig. 3.1.

In the next subsection, we will develop quantitative methods to choose a particular reconfiguration strategy out of all the possible ones by evaluating the performance of Petri nets.

3.2.3 Evaluating Alternative Reconfiguration Strategies

For the purpose of evaluating the performance of a Petri net, we replace the two times r_i' and r_i'' associated with a transition t_i with a single time r_i which is the expected time for the transition to occur.

We again turn to our example of Fig. 3.1, with the ESTG of Fig. 3.2. Fig. 3.9 shows the ESTGs corresponding to each of the three reconfiguration strategies. The Petri net corresponding to the first strategy has been given in Fig. 3.4. The Petri nets corresponding to the second and the third strategies are given in Fig. 3.10 and 3.11, respectively, with timing constraints.

We will choose the strategy with the maximum performance on the occurrence of the fault. In order to evaluate the performance of a Petri net on the occurance of a fault, the Petri net is augmented to form an

Error Petri net as follows:

1. Create a fault in one cycle of the Petri net be deliberately omitting the arc to the place where the token is lost.
2. Draw the further transitions that occur in the execution of the net, creating new lables for the place corresponding to the faulted condition, until a full cycle of the net has been executed, in a normal fashion.

Fig. 3.12, 3.13 and 3.14 are Error Petri nets of Fig. 3.4, 3.10 and 3.11, respectively. Since all these Petri nets are decision-free, we can apply the following results we have developed in performance evaluation [HO 79] to compare their performance.

Theorem: For a decision free Petri net, the minimum cycle time (maximum performance) C is given by

$$C = \max T_k/N_k: \quad k = 1,2,\ldots,q$$

where,

T_k = sum of the execution times of the transitions in circuits k

N_k = total number of tokens in the circuit k

q = total number of circuits in the net

In Fig. 3.12, considering the biggest loop, which, by inspection is $St_1At_2W't_7X't_2Mt_3Gt_4At_5Ft_6$, we can see that the maximum performance C_1 is

$$C_1 = r_1 + r_2 + r_7 + r_2 + r_3 + r_4 + r_5 + r_6$$

let $r = r_1 + r_2 + r_3 + r_4 + r_5 + r_6$, the maximum performance of the error free net. Then $C_1 = r + r_2 + r_7$ using $r_7' > r_3'' + r_4'' + r_5''$, we have in the limit,

$$C_1 = r + r_2 + r_3 + r_4 + r_5.$$

Similarly, we have in the limit

$$C_2 = 2r$$

$$C_3 = r + r_1 + r_2 + r_3.$$

Thus it can be seen that, strategies (1) and (3) are clearly superior to strategy (2). After comparing C_1 and C_3, we have strategy (1) superior to strategy (3) if $r_1 > r_4 + r_5$, and vice versa.

Therefore, we have developed a method which gives a very nice quantitative measure of the viability of the alternative, strictly from a throughput point of view.

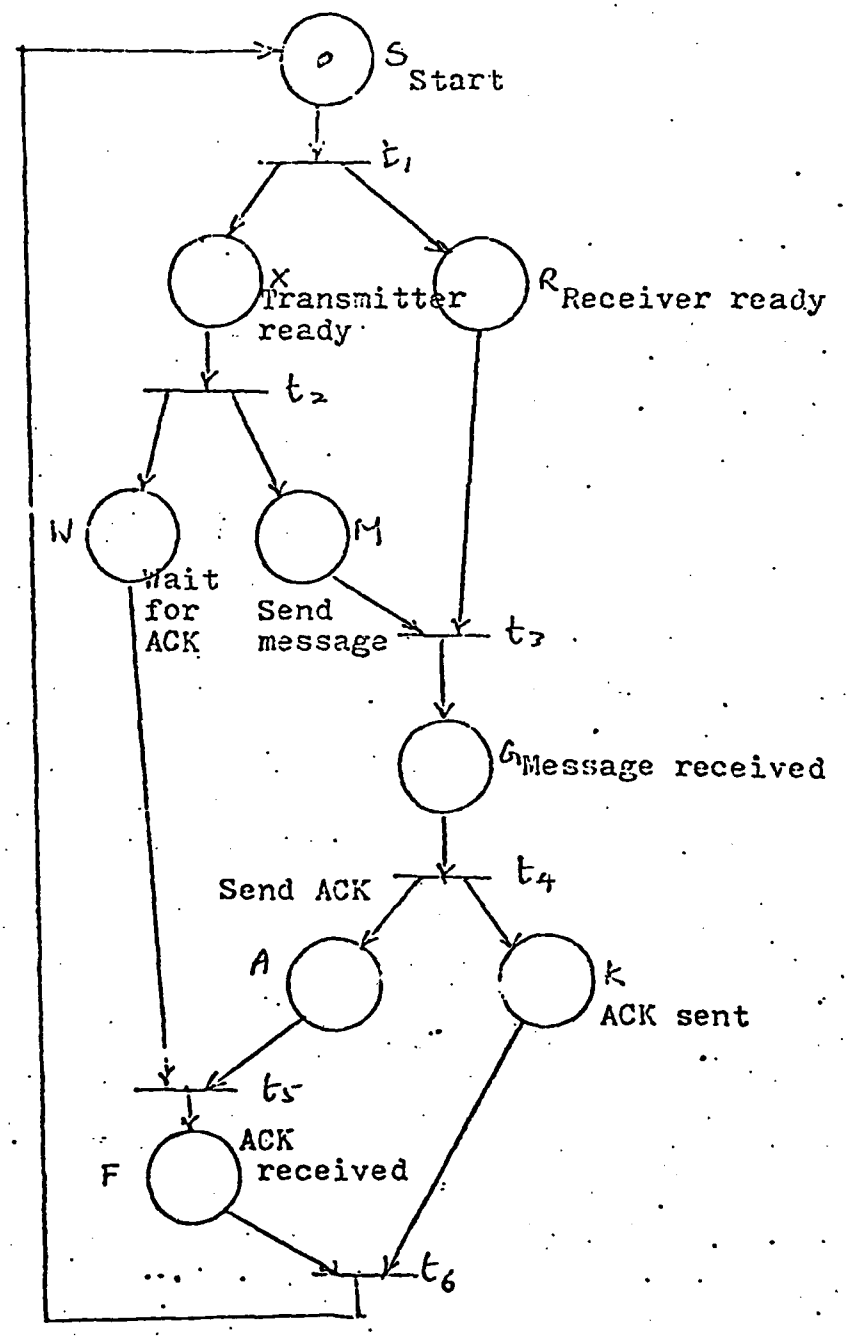


Figure 3.1 Petri net model of the exchange of messages

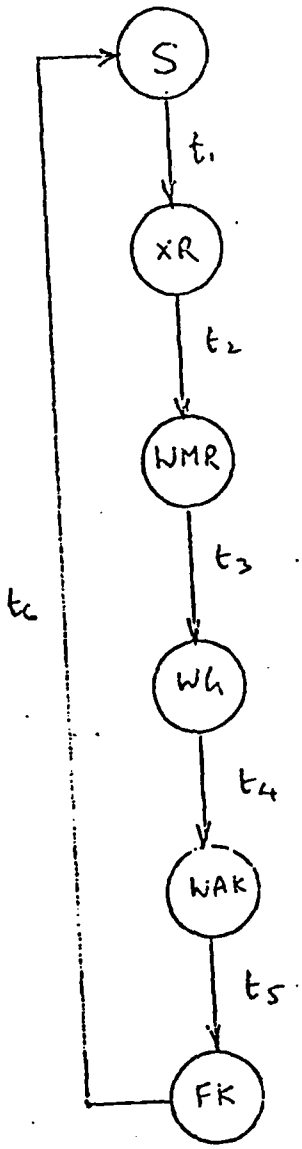


Figure 3.2 STG of figure 3.1

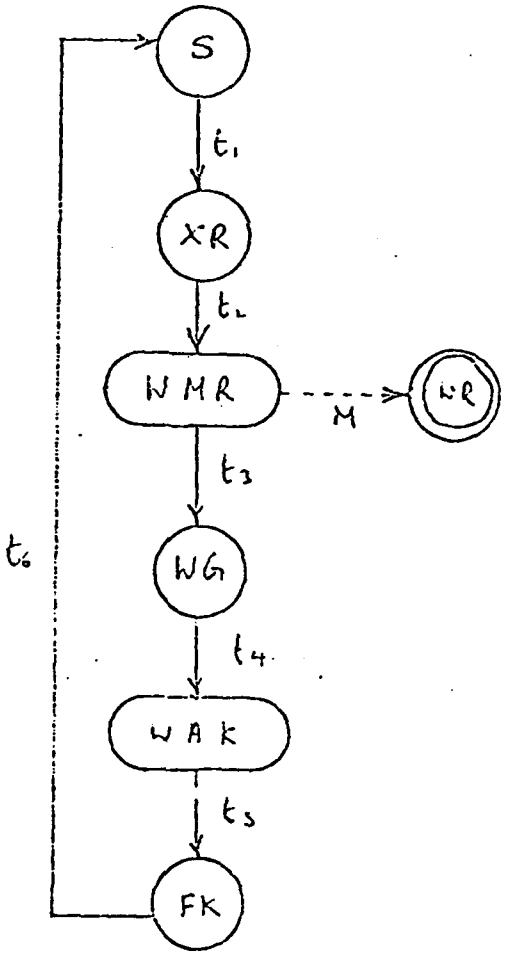


Figure 3.3 ESTG Of fig 3.1 for loss of token at M

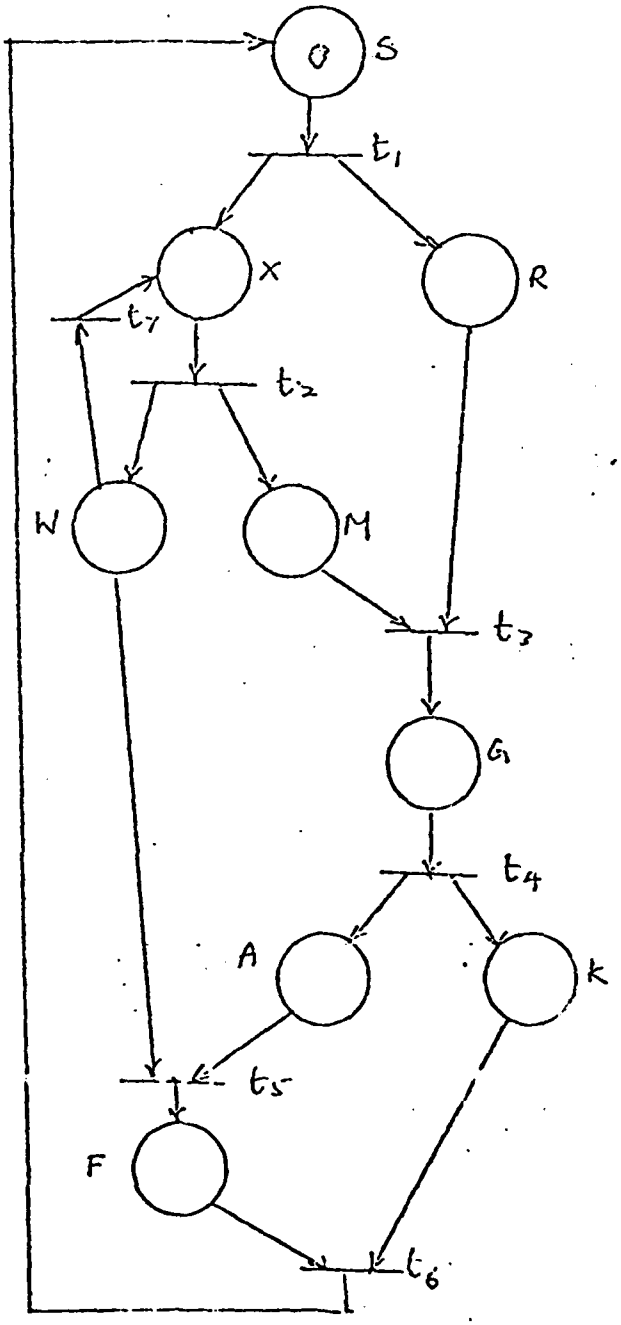


Figure 3.4 A possible reconfiguration strategy for fig 3.3

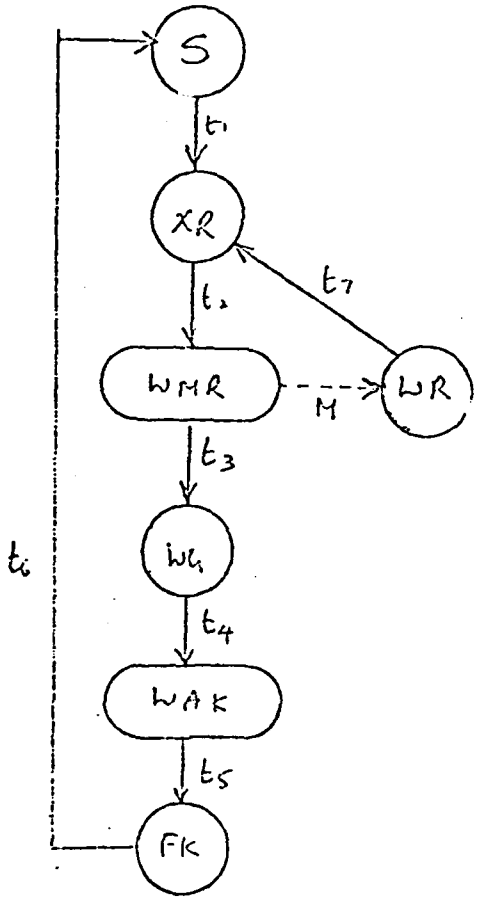


Figure 3.5 ESTG of fig 3.4

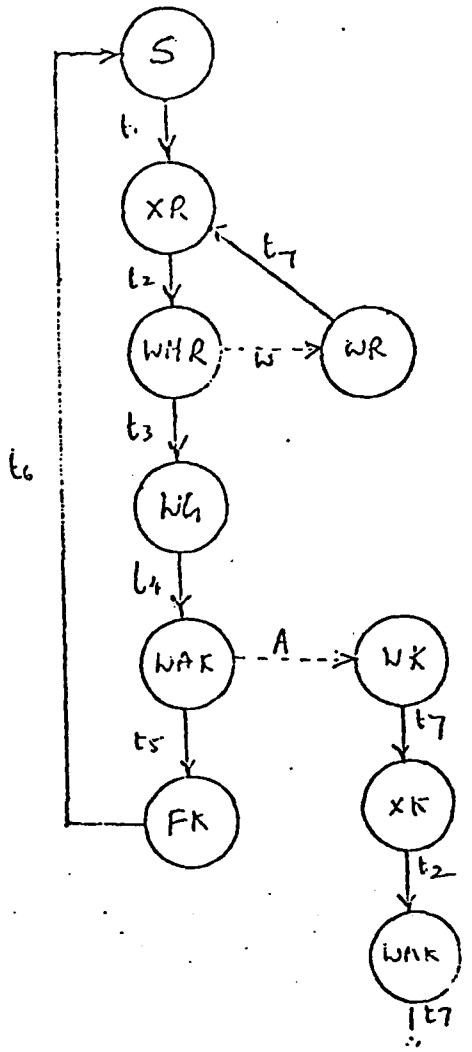


Figure 3.6 ESTG for the loss of token at A

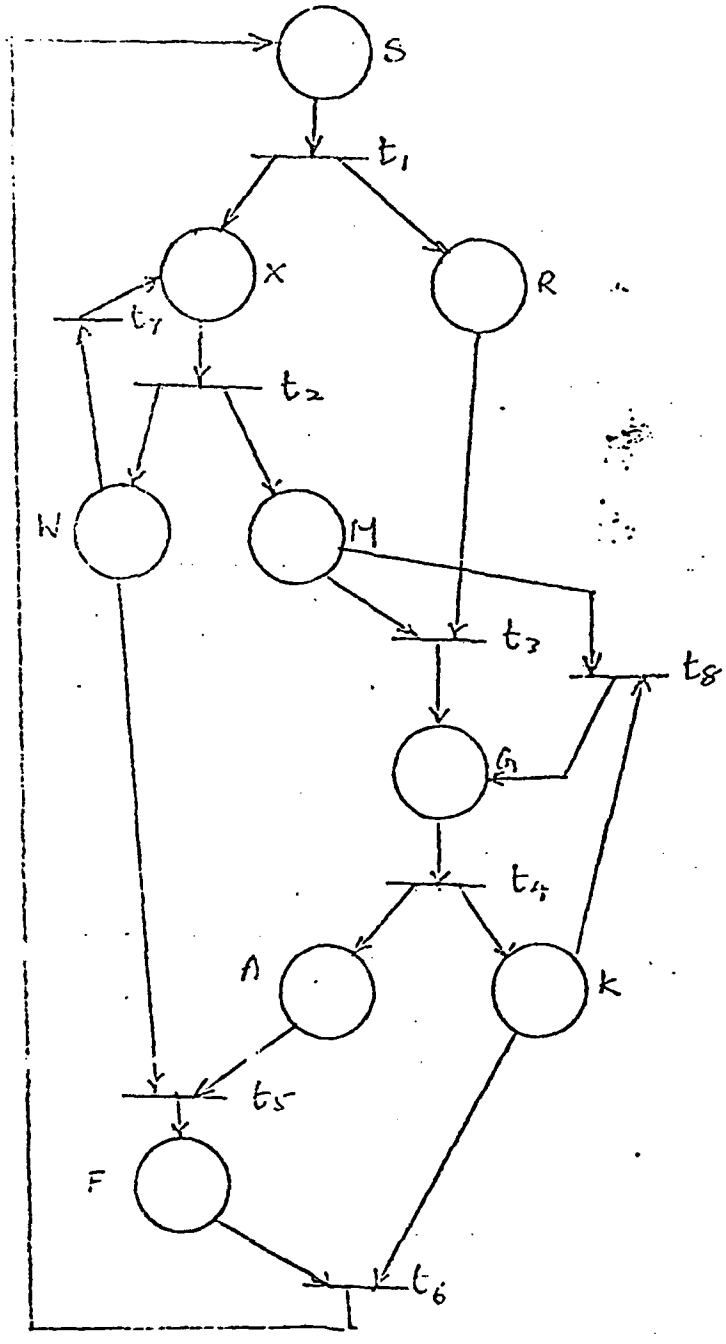


Figure 3.7 Petri net for the recovery of the failiure of fig 3.6

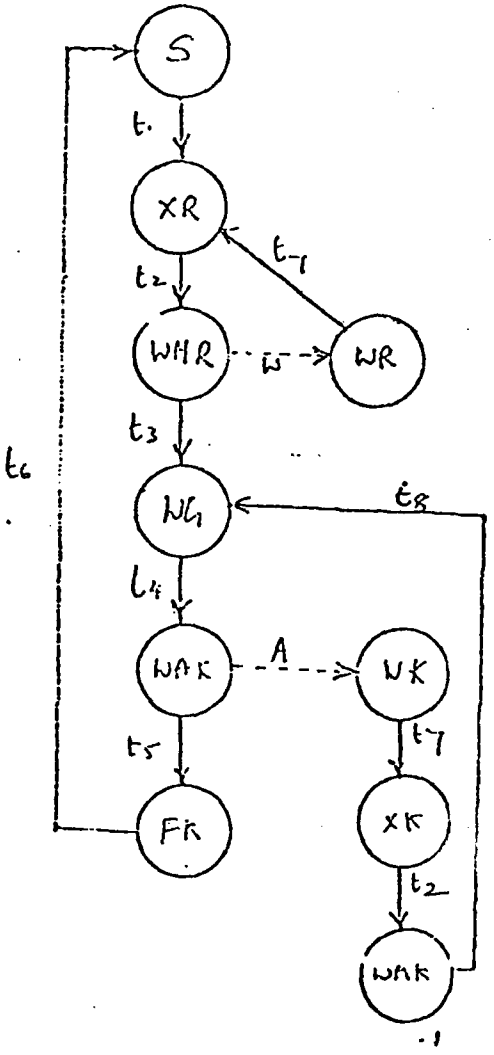


Figure 3.8 ESTG of fig 3.7

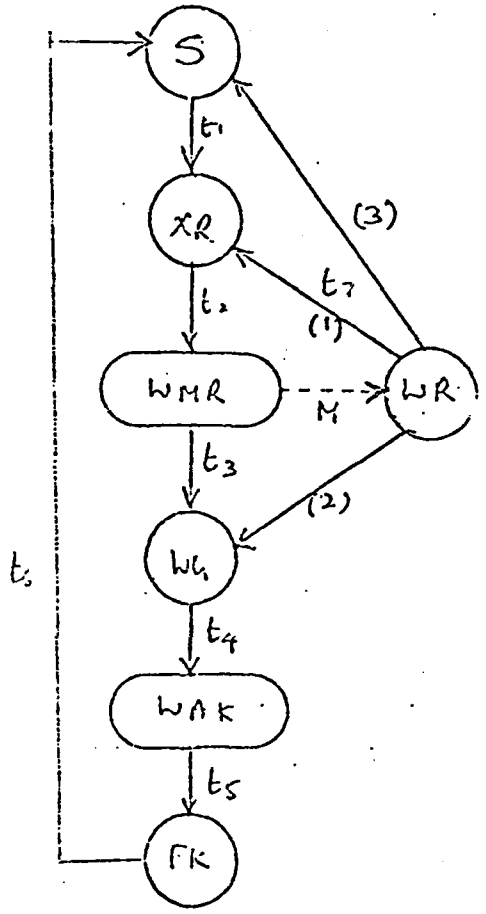


Figure 3.9 Three possible reconfiguration strategies for figure 3.3

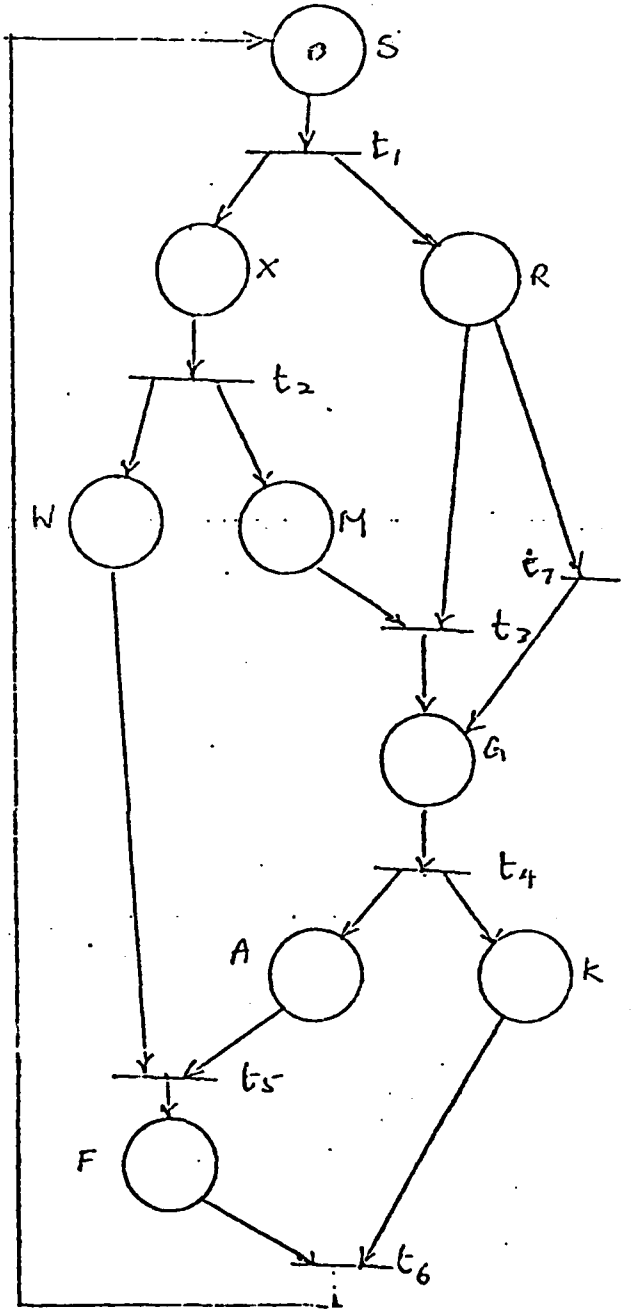


Figure 3.10 Petri net for strategy (2) with timing constraint $r_7' > r_2'' + r_3''$

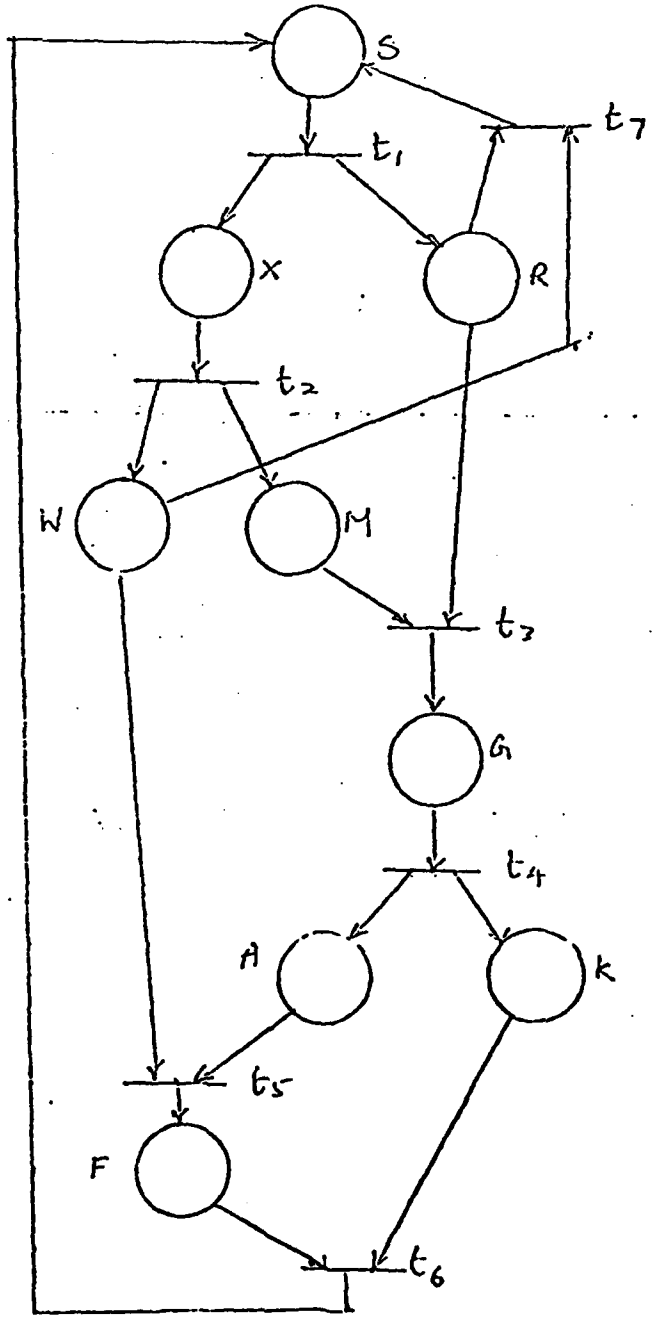


Figure 3.11 Petri net for strategy (3)
with timing constraint $r_7' > r_3''$

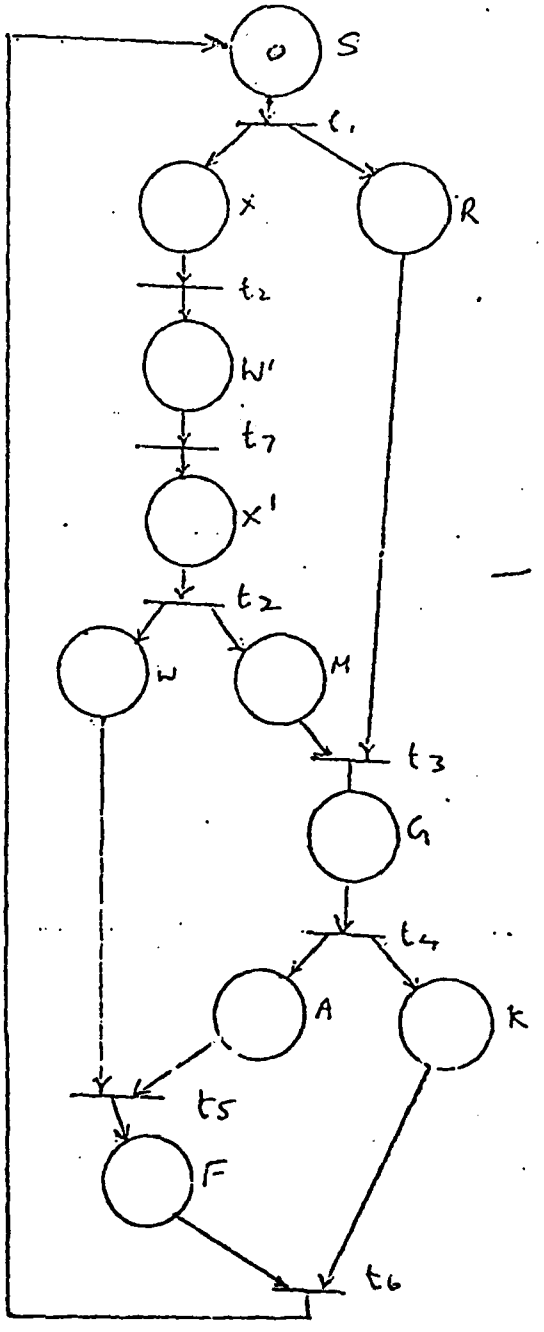


Figure 3.12 Error Petri net for figure 3.4

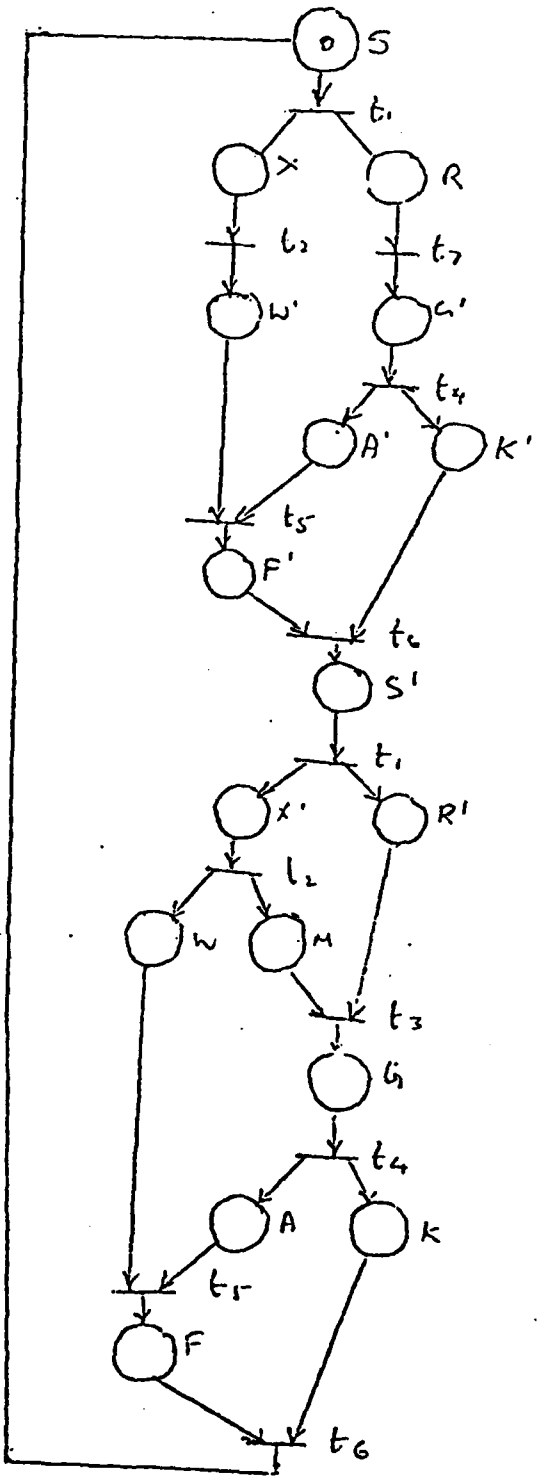


Figure 3.13 Error Petri net for figure 3.10

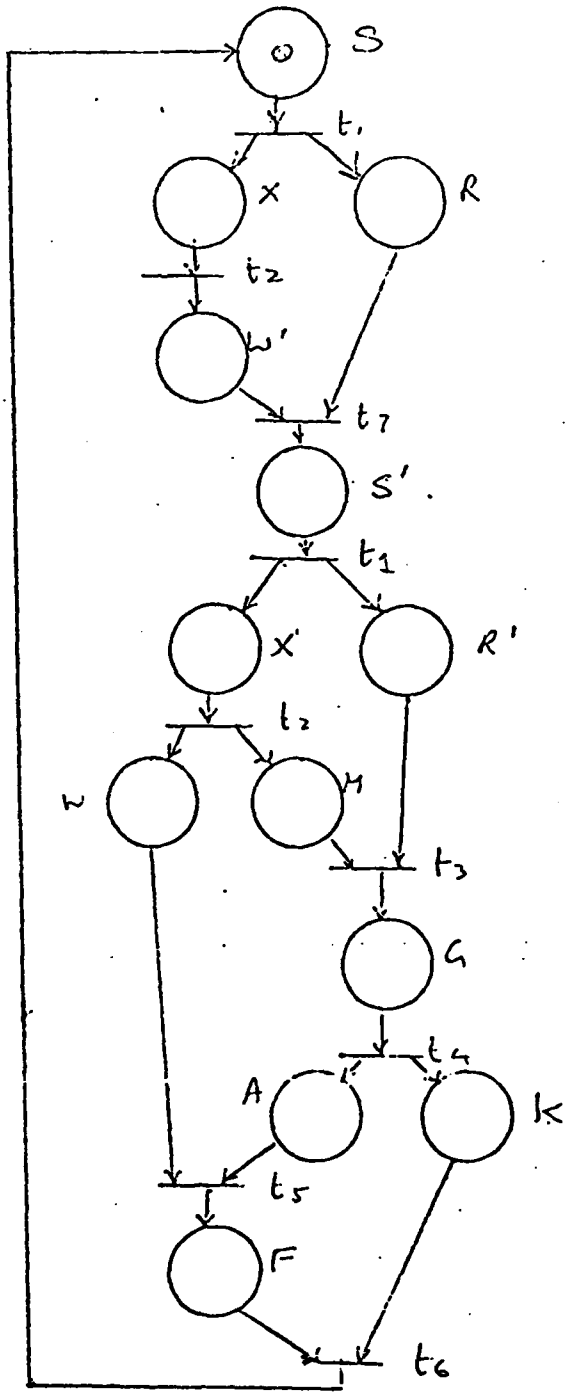


Figure 3.14 Error Petri net for figure 3.11

4. Classification of Interconnection Network

The interconnection network for the dynamic network has the following characteristics: The physical connections and the communication paths among nodes can both be varied with respect to time and spaces. We call this interconnection network "dynamic interconnection networks." Before we go into the detail design of the dynamic interconnection network for our problem, we first classified the existing interconnection networks in order to relate the similarities and distinquish the differences of the dynamic interconnection network to the other existing interconnection networks. The classification is based on the following attributes: the physical path, and the communication links.

		Path	
		Static	Dynamic
Connection	Static	unidirectional ring star, tree	Bidirectional ring ARPARNET
	Dynamic	cross bar Omega	unexplored (Dynamic interconnection network)

With this classification, we believe that the design of the dynamic interconnection network can be facilitated. For example, some techniques in using dynamic path from ARPARNET, and dynamic connection from Omega, etc can be borrowed to our problem.

5. Deadlock Detection

5.1 Past Work

Deadlock detection is performed by checking the resource allocation graph for cycles. This graph has nodes for transaction and resources. If a transaction is waiting to acquire a resource, an arc is drawn from the transaction to the resource. The arc direction is reversed if the transaction already holds the resource.

In a distributed environment, the resource allocation and deallocation centers may themselves be distributed. A centralized approach to detection is to cause these centers to send periodic reports to a common point where they are merged to build a single graph. However, because of message delays, all reports may not be equally up-to-date and false deadlocks may be detected. Ho [HO 79] has developed algorithms to deal with this problem, which require an additional time or space overhead compared to the centralized allocation situation.

Menasce and Muntz [MEN 79] have developed a truly distributed detection algorithm. However, this algorithm has the deficiency that several nodes may store information about a single allocation so that update costs are high when a deallocation occurs.

5.2 Work Accomplished

5.2.1 Centralized detection

The following may be stated to be a criterion to check if a cycle indicates a "true" deadlock:

Assertion 1: A cycle in a resource allocation graph built on the basis of reports from distributed allocation centers denotes a "true" deadlock if the arc leading into a node in the cycle is known to disappear only after the arc going away from it disappears.

This assertion may be proved by transitivity applied round the cycle.

Assertion 2: A two-phase discipline of acquiring and releasing resources will ensure the validity of the above criterion for process nodes.

By "two-phase" we mean that a transaction acquires all its resources before releasing any of them.

Since the criterion stated in assertion 1 is guaranteed for resource nodes by the lock controllers, we get the following result:

Consider an environment in which

a) resource allocation is distributed
b) transactions follow a 2-phase discipline in the use of resources
c) allocators send reports asynchronously to a central deadlock detector periodically.

Then, if G is a graph built on the basis of reports sent by the allocators at arbitrary time points, any cycle in G denotes a "true" deadlock.

Thus, if the above discipline is followed, no additional algorithmic complexity will be required for deadlock detection in the distributed situation as compared to the centralized allocation situation. Besides, the discipline may be enforced by other considerations e.g. serializability of transactions in database systems. Lastly, this algorithm will result in a sharp reduction in message traffic associated with the detection process compared to Ho's algorithms in situations where only a few of the nodes in the network control the allocation of the resources.

We have also studied two extensions of these concepts

a) rearrangement of the resource request and release points in a transaction to ensure a two-phase discipline and the tradeoffs involved.

b) adding information to resource requests to allow the detector to determine whether any two adjacent arcs in the graph satisfy the criterion given in Assertion 1.

5.2.2 Distributed Detection

We introduce some terms from Menasce and Muntz's paper [MEN 79].

TWF (K) is the portion of the transaction-wait-for graph at node K. This graph is a simplification of the resource allocation graph in that transaction dependencies are given directly rather than in terms of the resource through which one transaction depends on the other.

Sorig (T) is the site of origin of transaction T.

Blocking_Set (T) is the set of unblocked (active) transactions on which the transaction T is blocked, directly or indirectly.

Our algorithm may now be stated:

a) Say resource R cannot be granted to transaction T because it is held by T_1, T_2,..., T_R. Send $(T \rightarrow T_1)$,..., $(T \rightarrow T_R)$

   a_1) with instruction ADD to Sorig (T)

   a_2) with instruction CHECK to Sorig (T_1), Sorig (T_2),..., Sorig (T_R).

b) Say $(T \rightarrow T')$ is received at a node K

   b_1) If the instruction is ADD, then add $(T \rightarrow T')$ to TWF (K).

   b_2) If the instruction is CHECK, see if the addition of $(T \rightarrow T')$ would cause a cycle to form. If it would, appropriate action must be taken to break the deadlock. Further, if T' is blocked, then for all T" in Blocking_Set (T) send $(T \rightarrow T")$ to Sorig (T") with instruction CHECK.

This algorithm results in all stored dependencies being direct (instead of possible "condensations" as in [MEN 79]) and each dependency being stored only in one site (as against possibly multiple sites in [MEN 79]). However, more work needs to be done to take care of message delay problems like the ones described for the centralized detection case.

6. Scheduling

6.1 Introduction

The speed of digital circuits and components have been made faster than ever before through the advancement of technology. Yet there are a large number of applications which need processing power in excess of what is technology possible to provide. To overcome the speed limit of circuits and components, one approach is to segment the computable process into parallel processable segments and execute the program with a higher degree of concurrency. Recent advancement in LSI technology have provided low cost computing capability. The multi-processing approach appears to be more attractive to increase the speed of computing.

The multiprocessing of a set of tasks involved the following steps:

1. Detection and recognition of parallel processable tasks in a given computation.
2. Development of a schedule of sequencing the execution of the parallel tasks.
3. The problem of execution of parallel task sequences.

Here we are focusing only on step (2). We assume that step (1) has already provided the required input for step (2).

Although a lot of research has been done on the multiple processors scheduling problem, most of them do not consider the overhead effect that would happen in real environment because of coordination requirements of multi-processing. Direct measurements of a real network (ARPANENT) has been performed [KLE 74, COL 71]. It was shown (by accounting for processing, transmission and propagation times), that the expected round-up communication delay between two nodes in the ARPANET (SRI-UTAH) could easily be calculated to give 20 + 7.68b msec, where b is the length of the message in bits traveling this round trip. In this case

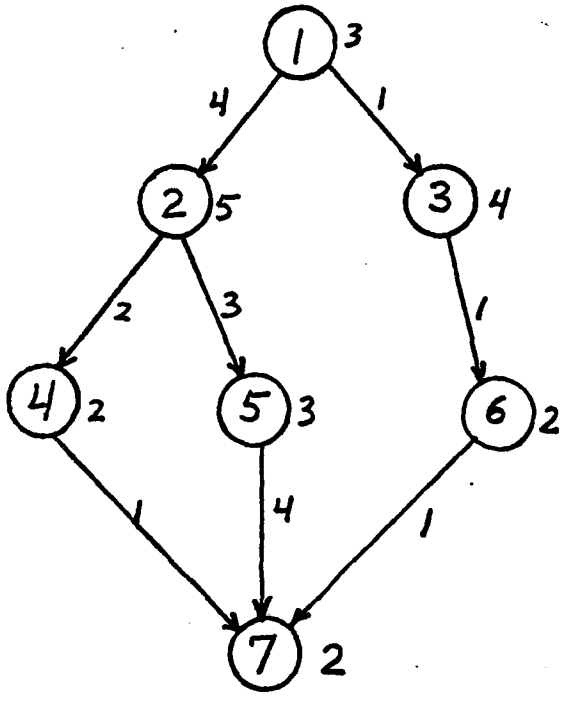
the delay time caused by communication overhead is by no means negligible. It should effect critically the scheduling strategy and ultimate efficiency of multi-processing. The objective of this report is to develop techniques that a program can be executed in the least time with the consideration of the overhead involved in parallel execution of tasks. The scheduling problems to be examined are expressed in terms of deterministic models. By this we mean that all the information required to express the characteristics of the problem is known before a solution to the problem (i.e. a schedule) is attempted.

6.2 Graph Model

Processor scheduling implies that jobs or tasks are to be assigned to a particular processor for execution at a particular time. Because many tasks or jobs can be candidates for execution, it is necessary to the relationships among jobs. A directed graph or precedence graph representation is probably the most popular representation in the scheduling literature. Fig. 6.1 shows one of several possible representations for a set of tasks. The nodes in this graph can represent independent operations or parts of a single program which are related to each other in time. By inspecting Fig.6.1 several pertinent observations can be made. First, the collection of nodes represents a set of tasks $T = \{T_1, T_2, \ldots, T_r\}$. The directed paths between nodes imply that a partial ordering or procedence relation $<$ exists between the tasks. Thus if $T_i < T_j$, task T_i must be completed before T_j can be initiated. Associated with each node is a second number which refers to the time required by a hypothetical processor to execute the code represented by the node. Besides indicating the precedence relationships among tasks, the edge, also have a number with it. This number represents the time delay needed to start the successor task if successor was assigned a different processor for

predecessors. If both tasks are assigned the same processor, then the time delay is 0 (i.e. the number with the edge need not be considered). The number of processors, of course, directly determines the amount of time required to executed the tasks in T. But, as will be discussed later, it is not necessarily true that execution time is inversly proportional to the number of processors. Sometimes because of large overhead, multi-processing seems to be worse than uniprocessing. Thus, given the task precedence graph and its associated node execution time and edge overhead, one could find optimal schedules for minimum time execution of the program with a specified number of processors.

NODE: TASK
DIRECTED EDGE: a. PRECEDENCE RELATIONSHIPS BETWEEN TASKS
b. OVERHEAD



Example:

Task 1 execution time = 3

Task 2 execution time = 5

Overhead between task 1 and 2 = 4

Figure 6.1-Graph model

7. Conclusion

In summary, we have developed design principles, algorithm concepts, feasibility criteria, and quantitative trade-offs in the following five areas: i) Routing Control and Relay Management; ii) Reconfiguration Control; iii) Interconnection Network Design; iv) Deadlock Detection; and v) Scheduling.

In relay management, we have devised a shortest path routing algorithm with expected execution time $O(\sqrt{n} \log n)$, where n is the number of nodes in the network. In reconfiguration control, we have investigated the optimal reconfiguration strategies and four different models for dynamic reconfigurable systems. Furthermore, we have developed a reconfiguration procedure by using Petri net models. We have also classified the existi g interconnection networks with respect to the characteristic of physical connections and communication paths in the networks.

In the area of deadlock detection, we have obtained the following results: i) in the case of centralized detection, we have evolved a discipline of requesting and releasing resources which obliviates the need to establish consistency between reports from different resource-controlling sites; ii) in the case of distributed detection, we have improved the algorithm given in [MEN 79] by reducing the message traffic required when a resource is released. In addition, we have used deterministic graph models to express the scheduling problem and to find the optimal schedules for the minimum execution time of the program with a specified number of processors.

The results we have obtained in our research provide guidelines and design laws for the systematic design and construction of distributed data processing systems such that the users' requirements are satisfied. By applying these results, we will be able to develop reliable, effective, modifiable networks with low costs and lead times.

8. References

[COL 71] Cole, G. C., "Computer network measurements: techniques and experiments," School of Engineering and Applied Science, University of California, Los Angeles, Engineering Report UCLA-ENG-7165, October 1971.

[HEL 78] Helvik, B. E., "An approach to optimal reconfiguration in dynamic fault-tolerant systems," The English Annual International Conference on Fault-Tolerant Computing, 1978.

[HO 79] Ho, G. S. "A systematic approach for the design and analysis of distributed computer systems," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1979.

[KLE 74] Kleinrock, L. and Naylor, W. E., "On measured behavior of the ARPA network," AFIPS Conference Proceedings, 1974 National Computer Conference 43, 767-780, 1974.

[KRI 79] Krishnam, K. V., "Design of reconfigurable distributed computer systems," Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1979.

[MA 80] Ma, Y. W., "A shortest path algorithm with average execution time $O(\sqrt{n} \log n)$," Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1980.

[MEN 79] Menasce, D. and Muntz, R., "Locking and deadlock detection in distributed databases," IEEE Transactions on Software Engineering, May 1979, pp. 195-202.

[TRO 77] Troy, R., "Dynamic reconfiguration: An algorithm and its efficiency evaluation," The Seventh Annual International Conference on Fault-Tolerant Computing, 1977.